Type-Based Compression of XML Data*

Christopher League
Long Island University
1 University Plaza, LLC 206
Brooklyn, NY 11201 USA
christopher.league@liu.edu

Kenjone Eng
Long Island University
700 Northern Blvd.
Brookville, NY 11548 USA
kenjone.eng@liu.edu

Abstract

The extensible markup language XML has become indispensable in many areas, but a significant disadvantage is its size: tagging a set of data increases the space needed to store it, the bandwidth needed to transmit it, and the time needed to parse it. We present a new compression technique based on the document type, expressed as a Relax NG schema. Assuming the sender and receiver agree in advance on the document type, conforming documents can be transmitted extremely compactly. On several data sets with high tag density this technique compresses better than other known XML-aware compressors, including those that consider the document type.

1. Motivation

In recent years, the extensible markup language XML [3] has become indispensable for web services, document markup, conduits between databases, application data formats, and in many other areas. Unfortunately, a significant disadvantage in some domains is that XML is extremely verbose. Although disk capacity is less often a concern these days, transmitting XML-tagged data still requires significantly more bandwidth and longer parse times (compared to a custom binary format, for example).

Compression of XML has been studied from a variety of perspectives. Some researchers aim to achieve minimal size [1, 4, 13], others focus on efficient streaming [9, 18] — a balance between bandwidth and encode/decode times — and still others answer XML queries directly from compressed representations [19]. Following Levene and Wood [12], we study how the document *type* or schema can be used to achieve smaller sizes.

Traditionally, one writes a document type definition (DTD) to constrain the sequencing of tags and attributes in an XML document. For most commonly-used XML formats, a DTD already exists. The DTD language is simple, but not terribly expressive, so a few competitors have arisen. We focus in particular on *Relax NG* by Clark and Murata [6]. It is expressive, but unlike *XML Schema,* it has a clean formal model [15] that is the foundation of our compression technique.

```
start = stm
stm = element seq { stm+ }
    | element assign { var, exp }
    | element print { exp* }
exp = element num { attribute val {text}, empty }
    | element id { var, empty }
    | element binop { attribute id {text}?, op, exp, exp }
    | element exp { attribute ref {text} }
var = attribute var {text}
op  = attribute op { "add" | "sub" | "mul" | "div" }
```

Figure 1: Relax NG schema in compact syntax. It defines the grammar for a simple sequential language – after Wang et al. [21] – expressed in XML.

```
<seq><assign var='x'><num val='7'/></assign>
     <print><binop op='add'>
         <binop id='r1' op='mul'><id var='x'/><id var='x'/></binop>
         <binop op='sub'><exp ref='r1'/><num val='5'/></binop>
     </binop></print>
</seq>
```

Figure 2: XML code representing a program in the sequential language of figure 1. When run, the program would output 93 (= 49 + 49 – 5).

Figure 1 contains a simple Relax NG schema written using the compact syntax. (There is an equivalent XML syntax that would make this example about 4 times longer.) This schema specifies the grammar for a small sequential language with variables, assignment, numbers, and arithmetic expressions. Note the use of regular expression operators (`+*?|`) and the `id`/`ref` attributes for reusing common sub-expressions.

Our original motivation for studying XML compression was to investigate the use of XML to represent abstract syntax trees and intermediate languages in compilers. The language in figure 1 is based on an example given by Wang et al. [21]. They developed an abstract syntax description language (ASDL) for specifying the grammars of languages, generating type definitions in multiple languages, and automatically marshaling data structures to an opaque but portable binary format. We hypothesize that XML and all its related libraries, languages, and tools could replace ASDL and prove useful elsewhere in the implementation of programming languages.

Figure 2 contains a small program in the sequential language, valid with respect to the Relax NG schema in figure 1. In conventional notation, it would be written as `x := 7; print [x*x + (x*x - 5)]`, but the common sub-expression `x*x` is explicitly shared via the `id`/`ref` mechanism. (One of the limitations of ASDL is that it could represent trees only, not graphs; however, managing sharing is essential for the efficient implementation of sophisticated intermediate languages [16].) These figures will serve as a running example in the remainder of the paper.

The main contribution of this paper is to describe and analyze a new technique for compressing XML documents that are *known* to conform to a given Relax NG schema. As a

simple example of what can be done with such knowledge, the schema of figure 1 *requires* that a conforming document begins with either <seq>, <assign>, or <print>; so we need at most *two bits* to tell us which it is. Similarly, the <binop> tag has an optional id attribute and a required op attribute. Here, we need just one bit to indicate the presence or absence of the id and two more bits to specify the arithmetic operator. (The data values are separated from the tree structure and compressed separately.) For this system to work, the sender and receiver must agree in advance on precisely the same schema. In that sense, the schema is like a shared key for encryption and decryption.

We are not the first to propose compressing XML relative to the document type – an analysis of related work appears in section 2 – but ours is the first such effort in the context of Relax NG, and in section 4 we report results that, on several data sets, improve upon other compression efforts that are known to us. The technique itself is detailed in section 3 and we close in section 5 by discussing limitations, consequences, and directions for future research.

2. Related work

A common theme among XML compressors is separating the tree structure from the text or data. Here we review several related techniques, paying close attention to the extent to which they use the schema or DTD. Some ignore it entirely, others optionally use it to optimize their operations, and some (like us) regard it as essential.

Girardot and Sundaresan [9] describe Millau, a modest extension of WAP binary XML format, that uses byte codes to represent XML tag names, attribute names, and some attribute values. Text nodes are compressed with a deflate/zlib algorithm [8] and the types of certain attribute values (integers, dates, etc.) are inferred and represented more compactly than as character data. Millau does not require a DTD, but if present it can be used to build and optimize the token dictionaries in advance.

Sundaresan and Moussa [18] build on this work, proposing *differential DTD compression,* which sounds similar in spirit to our idea. Unfortunately, the paper is short on implementation details, and they report poor run-time performance (of that particular technique) on all but the simplest sorts of schemas. Specifically, it was unable to compress Hamlet[1] in a reasonable amount of time; they were forced to abort the computation and omit that test case from the results.

Liefke and Suciu [13] implemented XMill. Its primary innovation compared to others mentioned here is to group related data items into containers that are compressed separately. To cite their example, "all <name> data items form one container, while all <phone> data items form a second container." This way, a general-purpose compressor such as deflate will find redundancy more easily. Moreover, custom semantic compressors may be applied to separate containers: date attributes, for example, could be converted from character data into a more compact representation. One of the main disadvantages of this approach is that it requires custom tuning for each data set to do really well.

Cheney's xmlppm [4] is an adaptation of the general-purpose *prediction by partial match* compression to XML documents. The path from root to leaf acts as a context for compressing the text nodes, which provides benefits similar to those of XMill's containers. Adiego

[1] from the Shakespeare XML corpus: http://www.ibiblio.org/bosak/xml/eg/shaks200.zip

et al. [1] augment this technique with a heuristic to combine certain context models; this yields better results on some large data sets. In our experiments, `xmlppm` was clearly the main contender. We expected Cheney's DTD-conscious compression techniques [5] to fare especially well, but as he reports, "for large data sets, `dtdppm` does not compress significantly better than `xmlppm`." In our tests, `dtdppm` never beat `xmlppm` by more than a few percent, and usually did worse; see section 4.

Toman [20] describes a compressor that dynamically infers a custom grammar for each document, rather than using a given DTD. There is a surface similarity with our technique in that a finite state automaton models the element structure, and is used to predict future transitions. His system was tested on three classes of data, but it never beat `xmlppm` on either the compression ratio or the run time.

Levene and Wood [12] propose a DTD-based encoding in much the same spirit as our work; however, there appears to be no implementation or experimental results. Instead, they prove optimality, but under the assumption that the DTD is non-recursive, which is not generally true for our target application area (representing intermediate languages).

Two newer efforts claim to beat `xmlppm` on some data sets by using information from the document type. Subramanian and Shankar [17] analyze DTDs to generate finite state automata and invoke arithmetic encoding at choice points. Harrusi et al. [10] describe a staged approach, where a DTD (which they call a dictionary) is converted to a context-free grammar for a specialized parser. Both of these appear similar in spirit to our approach, although we are the first to explore it in the context of Relax NG specifically. At press time, we have not yet obtained implementations of these competing techniques, but we look forward to performing direct comparisons in the future.

3. The technique

Our technique has been implemented in Java, as a tool called `rngzip` that supports much of the same command-line functionality as `gzip`. We benefitted greatly by using the Bali system by Kawaguchi,[2] a Relax NG validator and *validatelet compiler.* It builds a deterministic *tree automaton* [7, 15] from the specified schema. If also given an XML document, it checks whether the XML is accepted by the automaton. Alternatively, it can output a hard-coded validator for that particular schema in Java, C++, or C#.

We borrow the schema-parsing and automaton-building code from Bali. A diagram of the automaton induced from our sequential language schema is found in figure 3. The octagon-shaped states (1, 3, 8) are final/accept states. The transitions are labeled with XML tags or attribute predicates. The meanings of these predicates are described by example in the following table:

@id	contains an `id` attribute
!@id	does *not* contain an `id` attribute
!@~id	contains no attributes *except* possibly `id`
!@~(op| id)	contains no attributes except possibly `op` or `id`
!@*	contains no attributes at all

[2] `http://www.kohsuke.org/relaxng/bali/doc/`

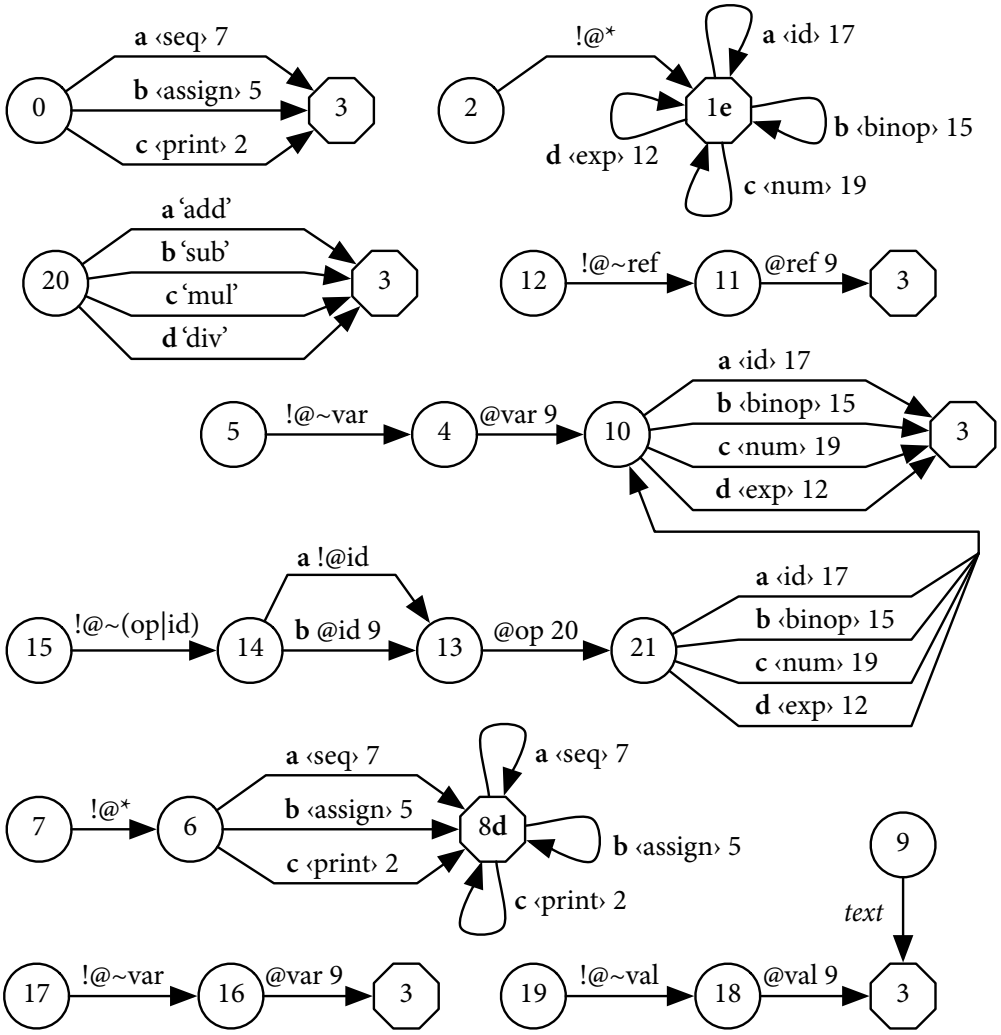

Figure 3: The tree automaton induced from the Relax NG schema in figure 1, and annotated with letters (a, b, c, . . .) at the choice points.

When the label of a transition is followed by a number, we jump to that state *as a subroutine* to validate the children of the current XML node (or the value of the current attribute).

Let's study how the program `<print><num val="42"/></print>` is validated by this automaton. Starting from state zero, there is a transition for `<print>`, so we push the target state 3 onto a stack and jump to state 2 as indicated by the transition. State 2 requires that the `<print>` just seen has no attributes. State 1 has a transition matching `<num>` so we push the target state 1 onto the stack and jump to 19. It requires that `<num>` has no attributes except `val`, and that `val` contains *text*. We pop back up to state 1. The close tag `</print>` is accepted in this final state, and we pop back up to 3. The end of the document is accepted in this final state, and (since the stack is now empty) the document is declared valid.

Given this automaton, a receiver can reconstruct an entire XML document by transmitting very little information. Whenever there is a *choice point* in the automaton, we just transmit *which* transition was taken. Whenever we encounter the *text* transition, we trans-

mit the matching text. Our program assigns unique labels to each outgoing transition from a choice point, shown in figure 3 as **a, b, c,** etc. The choice points in this automaton are states 0, 1, 6, 8, 10, 14, 20, and 21. Note that final states induce an additional choice: the automaton can either follow an outgoing transition, or stay and *accept*. This is why states 1 and 8 have a label *inside* the state.

So, in the case of the simple print statement above, we would just need to transmit **c, c,** `"42"`, **e**. Similarly, if we transmit **b,** `"x"`, **c,** `"8"`, the receiver ought to be able to reconstruct `<assign var="x"><num val="8"/></assign>`. Of course, it is even better than it looks: we are not transmitting the ASCII character **b;** since there are just 3 choices from state 0, we need only 2 bits to indicate which one, and 2 more bits for transition **c** from state 10. This is why it is critical that sender and receiver agree on precisely the same schema, and moreover that their implementations label transitions in the same order. (We had to impose a specific ordering of transitions on Bali, since it used a map based on object hash codes, which could vary between runs.)

Any white space outside of *text* nodes will be lost with this technique, but this is considered *ignorable* by the XML document model anyway, and it is trivial for the receiver to reformat the output if desired. In many cases, the decompressor will simply provide SAX events to some application, so the white space will not be missed; this is significantly more efficient than decompressing to a temporary file and then re-parsing it.

Now we step back to discuss other implementation details. Logically, the compressor outputs 3 distinct streams. The first is simply for configuration information: the URI of the schema, a checksum of the automaton, and an inventory of the other streams and how they are compressed. The second stream is for the bits that encode the tree structure, and the third is for the data values. We wrap the data stream with a generic compressor, such as Java's `GZIPOutputStream` (but any compatible filtering output stream can be easily substituted). Command-line options control whether and how each logical stream is compressed, and the results are compared in section 4.

Physically, these streams are combined into one by dumping each buffer as it becomes full, with a short header encoding the stream ID and block size. We call this a multiplexed stream and aimed initially to optimize it for efficient streaming, so that the memory requirements of both sender and receiver could be sub-linear in the total size of the document. Unfortunately, a particular property of Relax NG seems to conflict with the goal of streaming; see section 5.

In addition to the fixed-length encoding of transitions as bits, we implemented an adaptive Huffman encoding [11]. Using the fixed encoding, a choice point with 16 transitions will use 4 bits each, even if one transition is taken thousands of times and the rest are used sparingly, if at all. With the adaptive technique, the Huffman tree eventually encodes the more frequent transitions with proportionally shorter bit strings. Sometimes this technique beats the others, but it is not clear that it is worth the extra implementation complexity; this is also explored further in the next section.

4. Results

We compared `rngzip` with some other compressors – both generic and XML-specific – on several data sets; the results are in figure 4. In these experiments, the competition was `gzip`,

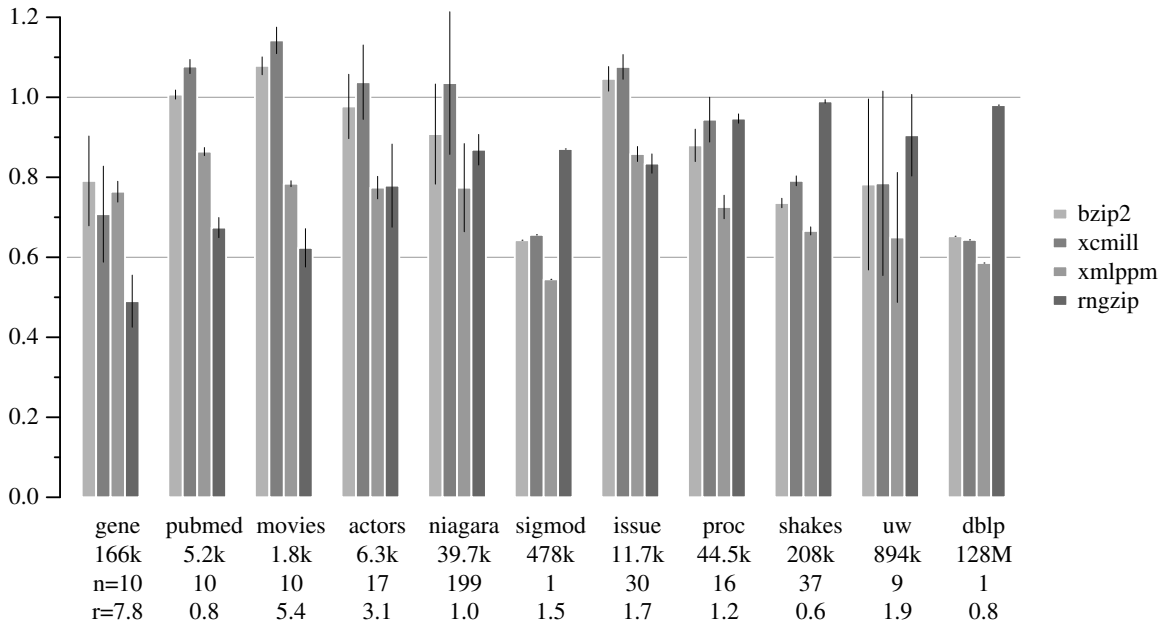

Figure 4: Effectiveness of several compressors over various data sets. The *y*-axis depicts compressed size, relative to gzip at 1.0. These are averages over many document instances, so error bars show one standard deviation above and below the mean. Beneath the data set labels are the average uncompressed sizes, the number of documents in the set (n), and the ratio of tags to text in the original (r, excluding ignorable spaces between tags).

`bzip2`, XMill version 0.8 [13], `xmlppm` 0.98.2 [4], and `dtdppm` 0.5.1 [5]. Unfortunately, this (alpha) version of `dtdppm` rarely beat `xmlppm` by more than a few percent, and on average did significantly worse. For this reason, we omitted its results from the graph.

The *gene* and *pubmed* data sets were from the NCBI.[3] Entrez Gene contains gene-specific information and focuses on the genomes that have been completely sequenced. We typed 'blood' into the search field and arbitrarily chose ten of the 3,315 results to save as XML files. PubMed includes citations from life science journals for biomedical articles. Similarly, we typed 'database' into the search box, and arbitrarily chose ten of the 80,032 records to save as XML files. The *movies* and *actors* came from IMDB, converted to XML by W4F.[4] *Niagara* is a collection of several other data sets – cars, clubs, company profiles, stock quotes, etc. – from the Niagara Query Engine.[5] Since the structure and results for these sets were fairly similar, we grouped them together. Likewise, we grouped course catalog data sets from the repositories at UW[6] and UBC.[7] DBLP provides bibliographic information on computer science journals and proceedings; this data set was also obtained from the UW repository. The sets *sigmod, issue,* and *proc* are various bibliographic data

[3] `http://www.ncbi.nlm.nih.gov/`

[4] `http://db.cis.upenn.edu/research/w4f.html`

[5] `http://www.cs.wisc.edu/niagara/data.html`

[6] `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`

[7] `http://www.cs.ubc.ca/nest/dbsl/xmlRep.html`

from ACM SIGMOD.[8] Finally, *shakes* is the previously mentioned Shakespeare corpus, including all the plays. Unfortunately, we were not able to find any real data sets using Relax NG natively, so the schemas we used were converted from DTD format by `trang`.[9]

To clarify the results shown in figure 4, the bars show the compressed size of each data set relative to the size produced by `gzip` (at 1.0). We found that `rngzip` performed exceptionally well – better than any other compressor we tried – for the *gene, pubmed,* and *movies* data sets. For the *actors* and *issue* data, `rngzip` is in the same neighborhood as `xmlppm`. Unfortunately, `rngzip` did not beat `xmlppm` on the remaining data sets. One factor in the effectiveness of our technique is the ratio (r) of tags to text in the XML source. The *gene* schema is almost ludicrously 'taggy.' Here is how a time-stamp is represented:

```
<Date><Date_std><Date-std>
      <Date-std_year>2006</Date-std_year>
      <Date-std_month>4</Date-std_month>
      <Date-std_day>7</Date-std_day>
      <Date-std_hour>18</Date-std_hour>
      <Date-std_minute>55</Date-std_minute>
      <Date-std_second>0</Date-std_second>
</Date-std></Date_std></Date>
```

We do particularly well on this because the long tag names are stored in the shared schema instead of the compressed output, and the tag structure is very regimented. On data sets such as *shakes* that consist more of lightly tagged text, our performance predictably degrades to that of `gzip`; we are after all using `GZIPOutputStream` to compress the data stream.

This ratio, however, is not the entire story. The *pubmed* set has a tag-to-text ratio in the same neighborhood as *shakes*, but `rngzip` performs far better on *pubmed.* This difference is explained by the structure of the schema. The tags in Shakespeare are somewhat few and repetitive (`speech speaker line line line`...) which is good for `gzip`, but there are technically many choices on every line (another line, end of speech, end of scene, end of act, ...) which is a disadvantage for `rngzip`. Nevertheless, it performs better than the differential DTD compression of Sundaresan and Moussa [18], which could not even compress Hamlet because of this property. Replacing our internal uses of `GZIPOutputStream` with `BZip2OutputStream`[10] would instantly make our tool more competitive on data sets such as *sigmod, shakes,* and *dblp* in terms of compressed sizes (at the cost of longer compression times). Incidentally, schemas representing compiler languages are more like *gene* and *pubmed* than Shakespeare, so we do expect `rngzip` to perform well in our target application area.

In section 3, we mentioned some variants, such as whether to use adaptive Huffman encoding, and how to compress each logical stream. For the results in figure 4, we used adaptive Huffman, and then applied `GZIPOutputStream` to both it and the data stream. This is the best approach on average, but in particular cases other settings perform slightly

[8] `http://www.acm.org/sigs/sigmod/record/xml/`

[9] `http://www.thaiopensource.com/relaxng/trang.html`

[10] `http://jakarta.apache.org/`

better. In our experiments, we found that the choice of tree encoder hardly made a difference; Huffman and fixed are nearly always equal in the end; the exception was for DBLP – our largest data set, at over 100 MB – where Huffman did make a significant difference. The effectiveness of compressing the bit stream after encoding is very situational; we found that it helps if the document is over 200 kb originally. Compressing the data stream is essential in every case.

5. Discussion and future directions

It is clear from the previous section that the effectiveness of `rngzip` across all the data sets is, at best, mixed. We have reason to believe, however, that the advantages of our technique on highly tagged, nested data are largely *orthogonal* to the benefits bestowed by `xmlppm` and XMill. This leaves open the hope of achieving the best of both worlds. In particular, the container expressions supported by XMill could be adopted wholesale and compressed separately into additional sub-streams of our multiplexed stream. We have even higher hopes for applying a context-sensitive PPM-based compressor (instead of gzip) to the data stream, but we leave this as future work.

We previously discussed several competing efforts on compressing XML data, but another piece of work deserves mention, from the domain of intermediate language representation. Amme et al. [2] encode programs using two innovative techniques: referential integrity and type separation. The effect is similar to what we do with XML, in that type-incorrect programs cannot be encoded at all. The compressed data are well-formed *by virtue of* the encoding.

In section 3, we mentioned a certain property of Relax NG that conflicts with the goal of streaming. Within an element specification, attributes and child elements can be intermixed; the attributes need not appear first. Here is an example where the element `<top>` has some 'big' content; imagine megabytes of data and enormous sub-trees.

```
top = element top { big, opt }
opt = attribute opt { text } | element opt { ... }
```

Following that, there is a *choice* between an `<opt>` element and an `opt` attribute (which applies to the parent element, `<top>`). What this means for decompressing is that we cannot output the `<top>` element until we know the result of the `opt` choice point, but that comes *after* the megabytes of data that must now be buffered. There is probably a workaround in the form of an automaton transformation that brings all the attribute transitions to the front. Martens et al. [14] provide formal evidence supporting this possibility: "[in the] tree grammars of Murata et al. [15]. . . every element in a document can be typed when its opening tag is met."

In addition to `text` and fixed strings used in figure 1, Relax NG can validate content using external libraries, such as the data type component of XML Schema. Our tool does not yet support any such libraries, but we hope in the future to use more compact encodings for content such as dates, *n*-bit integers, and base-64 binary data.

Another interesting extension – related to Bali's validatelet capability – is to generate the code for a compressor and decompressor specialized to a particular schema. This could be a real advantage for applications that save their documents and data in XML-based formats.

We hope that this line of work – on compact representations of XML – ultimately spells the end of the era of custom binary formats for storing or transmitting data.

Acknowledgment

We thank the anonymous referees for many enlightening comments on both the work and the presentation.

References

[1] J. Adiego, G. Navarro, and P. de la Fuente. Using structured contexts to compress semistructured text collections. *Information Processing and Management*, 2007.

[2] W. Amme, N. Dalton, J. von Ronne, and M. Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proc. Conf. on Programming Language Design and Implementation*. ACM, 2001.

[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. XML 1.1 (second edition). W3C recommendation, Aug. 2006.

[4] J. Cheney. Compressing XML with multiplexed hierarchical PPM models. In *Proc. Data Compression Conference*, 2001.

[5] J. Cheney. An empirical evaluation of simple DTD-conscious compression techniques. In *Eighth Int'l Workshop on the Web and Databases*, June 2005.

[6] J. Clark and M. Murata. Relax NG specification. OASIS Committee, Dec. 2001.

[7] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. 2002.

[8] P. Deutsch. DEFLATE compressed data format specification, version 1.3. RFC 1951, Aladdin Enterprises, May 1996.

[9] M. Girardot and N. Sundaresan. Millau: an encoding format for efficient representation and exchange of XML over the Web. *Computer Networks*, 33:747–765, 2000.

[10] S. Harrusi, A. Averbuch, and A. Yehudai. XML syntax conscious compression. In *Proc. Data Compression Conference*, pages 402–411. IEEE, 2006.

[11] D. E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6(2):163–180, June 1985.

[12] M. Levene and P. Wood. XML structure compression. In *Proc. 2nd Int'l Workshop on Web Dynamics*, 2002.

[13] H. Liefke and D. Suciu. XMill: an efficient compressor for XML data. In *Proc. ACM SIGMOD Conference*, pages 153–164. ACM, 2000.

[14] W. Martens, F. Neven, and T. Schwentick. Which XML schemas admit 1-pass preorder typing? In *Proc. Int'l Conf. on Database Theory*, volume 3363 of *LNCS*, pages 68–82. Springer, 2005.

[15] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Technology*, 5(4):660–704, Nov. 2005.

[16] Z. Shao, C. League, and S. Monnier. Implementing typed intermediate languages. In *Proc. Int'l Conf. Functional Programming*, pages 313–323. ACM, Sept. 1998.

[17] H. Subramanian and P. Shankar. Compressing XML documents using recursi[e finite state automata. In *Implementation and Application of Automata*, volume 3845 of *LNCS*, pages 282–293. Springer, 2006.

[18] N. Sundaresan and R. Moussa. Algorithms and programming models for efficient representation of XML for Internet applications. *Computer Networks*, 39:681–697, 2002.

[19] P. M. Tolani and J. R. Haritsa. XGRIND: A query-friendly XML compressor. In *Proc. Int'l. Conf. on Data Engineering (ICDE'02)*, pages 225–234. IEEE, 2002.

[20] V. Toman. Syntactical compression of XML data. In *Proc. Int'l Conf. on Advanced Information Systems Engineering*, 2004.

[21] D. C. Wang, A. W. Appel, J. L. Korn, and C. S. Serra. The Zephyr abstract syntax description language. In *USENIX Conf. on Domain-Specific Languages*, pages 213–227, 1997.